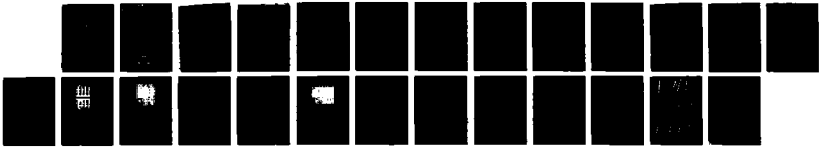
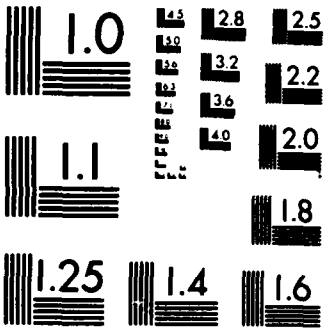
AD-A184 061 ARTIST: A SILICON ASSEMBLER FOR MESH ARRAYS(U) 1/1
PENNSYLVANIA STATE UNIV UNIVERSITY PARK DEPT OF
COMPUTER SCIENCE C S FUH ET AL. NOV 86 CS-86-31
UNCLASSIFIED ARO-20090.26-EL DAAG29-83-K-0126 F/G 9/1 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

2

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER ARO 20090.26-EL	2. GOVT ACCESSION NO. N/A	3. RECIPIENT'S CATALOG NUMBER N/A
4. TITLE *(and Subtitle)* ARTIST: A SILICON ASSEMBLER FOR MESH ARRAYS		5. TYPE OF REPORT & PERIOD COVERED Internal Technical
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR*(s)* C-S Fuh, K-Y Pun and R.M. Owens		8. CONTRACT/GRANT NUMBER(s) DAAG29-83-K-0126
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science The Pennsylvania State University University Park, PA 16802		10. PROGRAM WORK UNIT NUMBERS N/A
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE November 1986
		13. NUMBER OF PAGES - 22
14. MONITORING AGENCY NAME & ADDRESS N/A		15. SECURITY CLASS. *(of this report)* Unclassified

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the Abstract entered in Block 20)*
N/A

18. SUPPLEMENTARY NOTES
The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS

20. ABSTRACT

This paper describes a VLSI layout assembler, ARTIST, under development at Penn State. ARTIST performs transistor placement and interconnection within a module. Novel ideas used in the design of the assembler are described. A modular software design is used so that we can easily try different approximation algorithms for transistor placement. A comparison between simulated annealing and a totally random approach is presented. Surprisingly, the random approach is better for realistic running times. Finally, a hybrid approximation algorithms for transistor palcement is described and is shown to be better than either of the other two algorithms.

ARTIST: A Silicon Assembler for Mesh Arrays

Chiou Shann Fuh, Kong-Yee Pun, and Robert Michael Owens

CS-86-31 November 1986

Department of Computer Science
The Pennsylvania State University
University Park, PA 16802

ARTIST: A Silicon Assembler for Mesh Arrays

ABSTRACT

This paper describes a VLSI layout assembler, ARTIST, under development at Penn State. ARTIST performs transistor placement and interconnection within a module. Novel ideas used in the design of the assembler are described. A modular software design is used so that we can easily try different approximation algorithms for transistor placement. A comparison between simulated annealing and a totally random approach is presented. Surprisingly, the random approach is better for realistic running times. Finally, a hybrid approximation algorithm for transistor placement is described and is shown to be better than either of the other two algorithms.

CATEGORIES

4., 11., 3., 8.

INTRODUCTION

ARTIST is a tool under development at Penn State which generates a layout, in CMOS mesh array form [Be], for a module from its formal description. The name ARTIST, rather than the sometimes overused name "silicon compiler," was chosen due to the tightness between the formal description and the layout generated by ARTIST from that description. ARTIST is a key part of a CAD system under development at Penn State [OI1]. Other tools in this CAD system include: LOGICIAN, a tool for module generation which performs multi-level logic reduction [BOI]; COMPOSER, a tool for module placement within the target architecture; SIMULATE, a simulation tool [OI2]; and V, a layout verification tool [RI].

ARTIST owes its existence to both pragmatic and academic reasons. The academic reason was to provide a test bed so that different approximation algorithms for efficiently finding near optimal layouts could be tried. The pragmatic reason was the growing need to be able to quickly generate layouts for some of the VLSI architecture projects ongoing at Penn State.

The part of our design system surrounding ARTIST is shown in Figure 1. This paper first introduces the language in which the formal description of the module can be specified. ARTIST actually accepts only a restricted version of the full language. Thus, a Parser is used to transform a description which uses the full language into a description using only the language subset. LOGICIAN, the multi-level logic minimization tool which feeds ARTIST as shown in Figure 1, outputs the module description using this language subset. The layout program, ARTIST, which performs transistor placement and interconnect is then discussed. Finally, several layout optimization algorithms and their performance are presented.

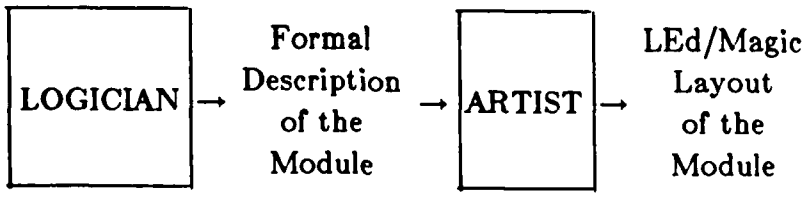


Figure 1. Portion of Design System Surrounding ARTIST

THE LANGUAGE

We wanted the form used to describe the layout to be generated by ARTIST to satisfy the following.

1) The form should allow a close relationship between a formal description of a module and the layout generated from that description.
2) The form would not in itself restrict the range of layouts which can be generated.
3) Descriptions expressed in other forms (i.e., net list) could be easily translated into descriptions expressed in our form.

To achieve these goals, a procedural based language was developed. We will briefly describe the language in this paper. A full description can be found in [Ow].

The description of a module consists of a set of procedures. Each of the procedures consists of a set of statements. The primary statement of the language is the assignment statement. Like the assignment statement of other languages, the assignment statement of our language is used to supply information to determine the value that a given variable has at any given time. However, the assignment statement of our language is quite different from the assignment statement of other languages in several important ways. The syntax of the assignment statement of our language which is given by

$$< e > a = b;$$

where e is a boolean expression and a and b are either variables or boolean constants (1 for true and 0 for false) is different. A boolean expression is an expression formed using only & (and), | (or), and ! (negation/not). The expression e is the assertion part and the equality $a = b$ is the equality part of the statement. An assignment statement is interpreted in the following way: if e is true, then a and b have the same value. An assignment statement for which the assertion part is true is said to be in an active state. Also, semantics of the assignment of our language which is given by the following is different.

1) The symbol = in the equality part implies equality, not normal assignment; that is, either variable can be changed so as to make the equality true. The symbol = was chosen because of the lack of the symbol ≡ in most text entry character sets.
2) The order of the statements within a procedure has no bearing on the manner in which they are interpreted including the order in which they

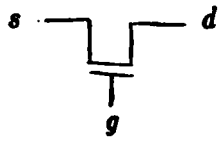
are executed.

3) The statements are not executed as the term normally applies. If the assertion part of the statement is true, the equality part is made and kept true by possibly changing the value of either variable as long as the assertion part is true. If this condition cannot be obtained, an error is assumed.

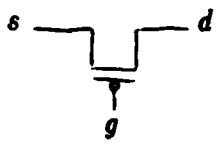
4) Unless it is necessary to change the value of a variable to either directly or indirectly satisfy the equality part of an active assignment statement, the value of a variable is assumed to remain unchanged.

There are two primitive forms of the assignment statement as shown below.

$< g > s = d;$ $< !g > s = d;$



N channel



P channel

where s, d, and g are variables and ! is boolean negation (not). Note that these two primitive forms correspond to a N channel and a P channel field effect transistor.

Assignment statements can be manipulated in the following ways. The statements

$< e_1 > a = b;$ $< e_2 > a = b;$

infer the statement

$< e_1 \mid e_2 > a = b;$

and vice-versa. Also, if b is not otherwise used, the statements

$< e_1 > a = b;$ $< e_2 > b = c;$

infer the statement

$< e_1 \mathcal{B} e_2 > a = c;$

and vice-versa.

Besides the assignment statement, our language also supports a procedure call statement. The effect of using a procedure call is that normally associated with a textual macro call or an Algol procedure call. Thus, the effect of a procedure call is that obtained by replacing the procedure call by the body of the procedure where formal parameters have been renamed to their corresponding actual parameters and other variables renamed as necessary.

The language has also been extended to support loop statements (for which the range is known). conditional statements (for which the condition can be computed at compile time), and array variables (bit vectors). To more fully illustrate the language, the complete formal description of a typical module of modest size, the mcell [OI1, IO], which performs a restricted base 4 signed digit multiplication, is given in Figure 2.

$$
\begin{array}{l}
\mathit{mcell}\,(m\,0,\ m\,1,\ m\,2,\ x\,0,\ x\,1,\ x\,2,\ r\,0,\ r\,1,\ r\,2,\ q\,0,\ q\,1) \\
\{ \\
\qquad < x\,0\ \&\ r\,0 > t\,0 = 0; \\
\qquad < !x\,0 \mid !r\,0 > t\,0 = 1; \\
\qquad < t\,0\ \&\ q\,0 > m\,0 = 0; \\
\qquad < !t\,0\ \&\ !q\,1 > m\,0 = 1; \\
\qquad < r\,1\ \&\ x\,0 > t\,1 = 0; \\
\qquad < !r\,1 \mid !x\,0 > t\,1 = 1; \\
\qquad < r\,0\ \&\ x\,1 > t\,2 = 0; \\
\qquad < !r\,0 \mid !x\,1 > t\,2 = 1; \\
\qquad < t\,1 > t\,3 = 0; \\
\qquad < !t\,1 > t\,3 = 1; \\
\qquad < t\,2 > t\,4 = 0; \\
\qquad < !t\,2 > t\,4 = 1; \\
\qquad < (t\,1 \mid t\,4)\ \&\ (t\,2 \mid t\,3)\ \&\ q\,0 > m\,1 = 0; \\
\qquad < ((!t\,1\ \&\ !t\,4) \mid (!t\,2\ \&\ !t\,3))\ \&\ !q\,1 > m\,1 = 1; \\
\qquad < (x\,0 \mid x\,1)\ \&\ r\,2 > t\,5 = 0; \\
\qquad < (!x\,0\ \&\ !x\,1) \mid !r\,2 > t\,5 = 1; \\
\qquad < (r\,0 \mid r\,1)\ \&\ x\,2 > t\,6 = 0; \\
\qquad < (!r\,0\ \&\ !r\,1) \mid !x\,2 > t\,6 = 1; \\
\qquad < (t\,5 \mid x\,2)\ \&\ (t\,6 \mid r\,2)\ \&\ q\,0 > m\,2 = 0; \\
\qquad < ((!t\,5\ \&\ !x\,2) \mid (!t\,6\ \&\ !r\,2))\ \&\ !q\,1 > m\,2 = 1; \\
\}
\end{array}
$$

Figure 2. Description for mcell

THE PARSER

To reduce the overall complexity of ARTIST and for reasons which will become clear later, it was decided that ARTIST itself would only accept a semantic and syntactic subset of our language. Briefly, ARTIST itself does not support procedure calls (or any other of the extended features of the language). Furthermore, for each *defined* variable, a, ARTIST requires exactly two consecutive statements of the form

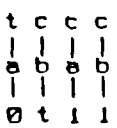
$$< e_d > a = 0;$$
$$< e_u > a = 1;$$

Furthermore, variable a may otherwise be used only in the assertion part of an assignment statement. Variables which are not defined variables, referred to as *used* variables, may only appear in the assertion part of an assignment statement. Boolean negation may not be used in expressions e_d and e_u except that all the variables of e_u must be negated. Note that the description of mcell given in Figure 2 conforms to these restrictions.

For the most part, having ARTIST accept only a subset of the language is not much of a problem for layout descriptions generated by our other CAD tools (e.g., LOGICIAN [BOI]). It is fairly easy to write these tools so that they produce a description using only the subset. However, for human generated input or for input translated from a graphic form (i.e., schematic capture), very often the conditions imposed by the subset seem unnatural and unduly restrictive. To get around this problem, a standard full language to sub language translator, Parser, was developed.

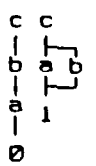
As illustrated in the previous section, a statement can be manipulated using a fairly simple set of rules. For example, it is fairly obvious that the set of statements

$$< a > t = 0;$$
$$< b > c = t;$$
$$< !a > c = 1;$$
$$< !b > c = 1;$$



can be transformed into

$$< a \ \& \ b > c = 0;$$
$$< !a \ | \ !b > c = 1;$$

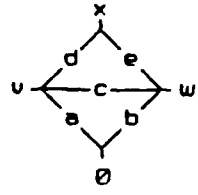


assuming that variable t is otherwise unused. Note the transformed set of statements is

acceptable to ARTIST while the original set is not. At first, it would seem that the tasks of Parser can be trivially performed. However, this is not always the case.

The first nontrivial situation encountered by Parser is caused by the use of shared terms caused by bidirectional paths. For example, consider the following set of statements.

$$
\begin{aligned}
&< a > v = 0;\\
&< b > w = 0;\\
&< c > v = w;\\
&< d > x = v;\\
&< e > x = w;
\end{aligned}
$$

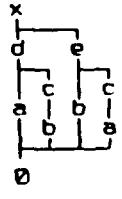


Translating these statements to one of the form

$$< f > x = 0;$$

cannot be performed by a simple pairwise serial or parallel combining of the original set of statements. Parser handles this case by first replicating the bidirectional path (e.g., as implied by $v = w$) to produce the following set of statements.

$$
\begin{aligned}
&< a > v = 0;\\
&< b > u = 0;\\
&< c > v = u;\\
&< d > x = v;\\
&< b > w = 0;\\
&< a > y = 0;\\
&< c > w = y;\\
&< e > x = w;
\end{aligned}
$$



These statements can now be translated using normal series and parallel combinations to produce

$$< d \ \& \ (a \mid c \ \& \ b) \mid e \ \& \ (b \mid c \ \& \ a) > x = 0;$$

assuming that variables v, u, w, and y are otherwise unused. An important implication of Parser's elimination of shared terms is that descriptions which contain "pass" transistors are converted to logically equivalent descriptions which do not.

The second nontrivial situation encountered by the Parser is caused by the use of boolean negation. The original language allows the free use of negation in the assertion. However, the subset accepted by ARTIST greatly restricts the use of negation. Parser handles this case by first pushing the use of negation to the literals of the expression. For example,

$$< !(a \ \& \ (!b \ | \ !c \ ))> x \ = 0;$$

is translated into

$$< !a \ | \ b \ \& \ c > x \ = 0;$$

Illegal variable negation is then corrected by introducing a dummy variable. Thus, the statement

$$< !a \ | \ b \ \& \ c > x \ = 0;$$

is translated into

$$< d \ | \ b \ \& \ c > x \ = 0;$$
$$< a \ > d \ = 0;$$
$$< !a > d \ = 1;$$

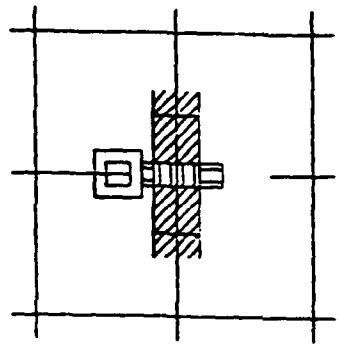
The effect of this transformation is that Parser introduces an inverter to generate the needed form (inverted/noninverted) of a variable. Parser does not attempt to optimize translation of illegal negations. Optimization is provided by other tools (i.e., LOGICIAN).

MESH ARRAYS

Mesh arrays were originally conceived as a structured aid to the hand layout of multi-level CMOS logic [Be]. It is surprisingly simple and efficient to create the layout for a mesh array by modifying a single general template. Mesh arrays are, at present, implemented using a two level metal CMOS process as supported by MOSIS. Physically, horizontal first level metal segments are used for transistor interconnections, power, and ground. Hence, logically, row segments of the mesh are allocated to variables. Physically, consecutive vertical diffusion segments are used to form the pullup (pull the gate's output toward Vdd) and pulldown (pull the gate's output toward Gnd) part of

each gate. Hence, logically, columns of the mesh are allocated to statements. Vertical second level metal segments are used to connect the pullup and pulldown part of each gate and to distribute module inputs and outputs.

The physical structure of mesh arrays can be described in more detail through a constructive description. The mesh array for a circuit consisting of a single N channel transistor is illustrated in Figure 3.

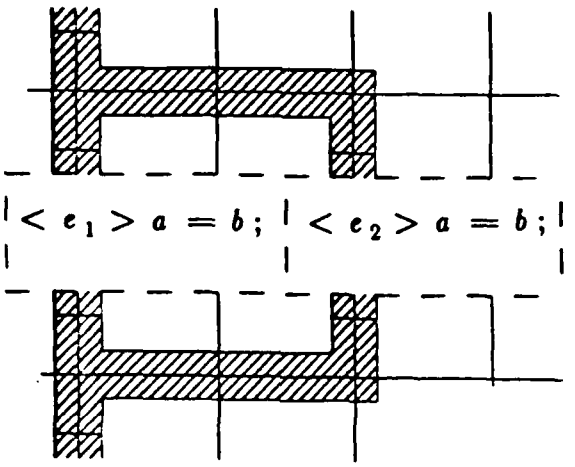


$< g > s = d$

Figure 3. Single Transistor Mesh Array

A horizontal first level metal segment, which is carrying the signal associated with variable g, is connected to the gate (via a polycontact) of the transistor.

The mesh array for a circuit consisting of a parallel connection of two subcircuits is illustrated in Figure 4.

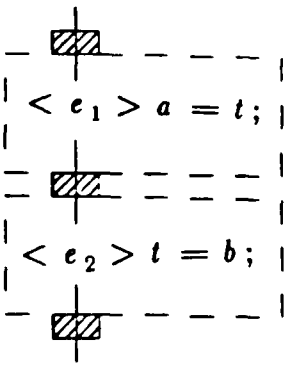


$< e_1 \mid e_2 > a = b;$

Figure 4. Parallel Connected Subcircuits

Since the two subcircuits share the same well, they both must consist of the interconnection of only N channel (P well) transistors or of only P channel (N well) transistors. Note that, because of conflicts in the allocation of row segments, both subcircuits may have to be stretched in the vertical direction.

Likewise, the mesh array for a circuit consisting of a series connection of two subcircuits is illustrated in Figure 5.



$< e_1 \ \& \ e_2 > a = b;$

Figure 5. Series Connected Subcircuits

Again since the two subcircuits share the same well, they both must consists of the in-

terconnection of only N channel (P well) transistors or of only P channel (N well) transistors.

Finally, a gate consists of the interconnection of two subcircuits is illustrated in Figure 6.

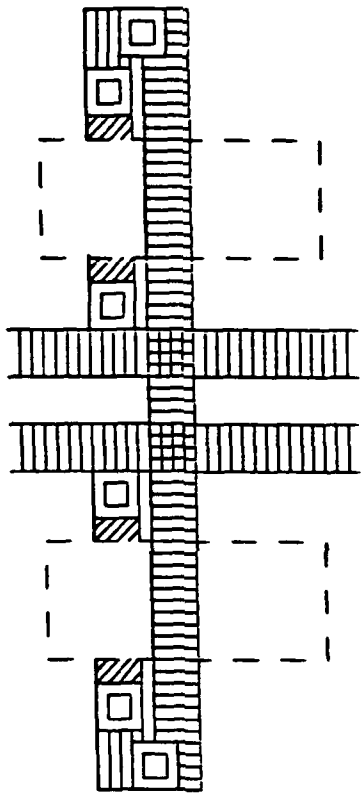
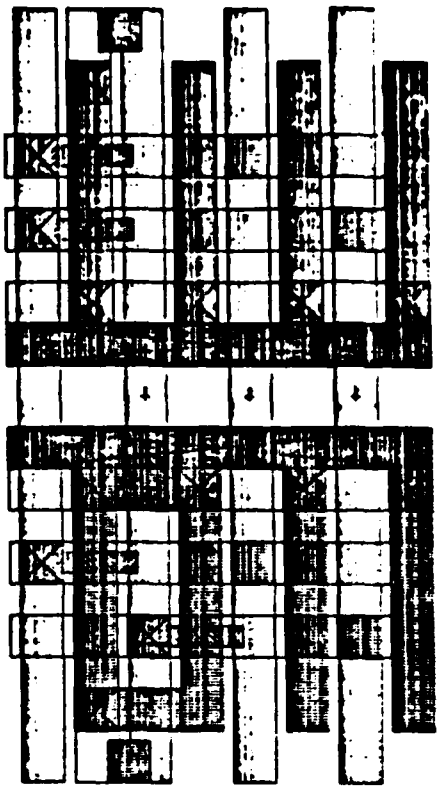


Figure 6. Single Gate

The pullup circuit must consist of only P channel (N well) transistors and the pulldown circuit must consist of only N channel (P well) transistors. To complete our description, the mesh array for a two input nor gate is illustrated in Figure 7.



```
nor (a , b , c )
{
        < a  |  b  > c  = 0;
        < !a  &  !b  > c  = 1;
}
```

Figure 7. Two Input Nor Gate

Now that the physical structure of mesh arrays has been described, the reasons behind the restricted language accepted by ARTIST should be apprent. The two statements associated with a defined variable describe the pullup and pulldown part of the CMOS gate which generates that variable. The pullup part of the gate specified by e_u contains only P channel transistors and the pulldown part of the gate specified by e_d contains only N channel transistors. Furthermore, only literals of e_u and e_d can be negated. However, a variable which is negated, which would imply a P channel transistor, cannot appear in e_d. Also, a variable which is not negated, which would imply a N channel transistor, cannot appear in e_u.

Mesh arrays do not structurely support pass transistors. Hence, bidirectional paths are not permitted. Defined variables represent the outputs of the gates of the module and used variables represent inputs supplied to the module. Figure 8 illustrates a hand generated mesh array layout for the formal description of the mcell module given in Figure 2.

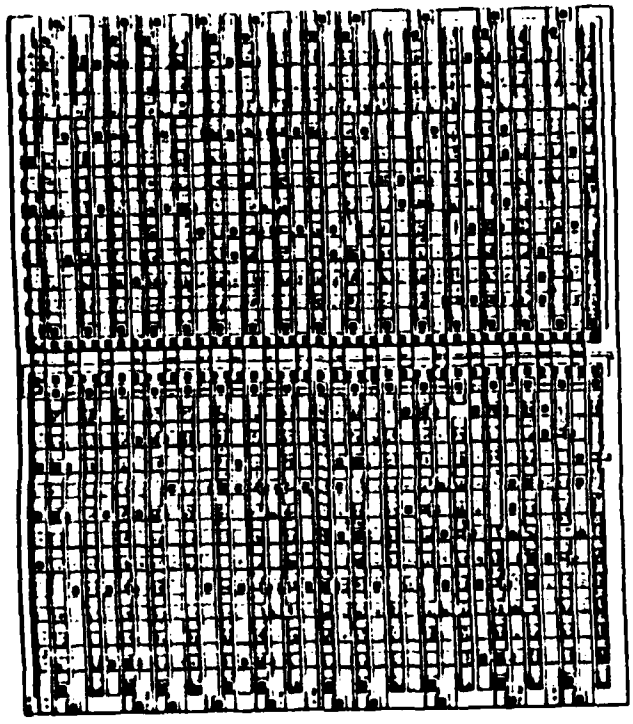


Figure 8. Hand Generated mcell Layout

This layout is seventeen columns by twenty four rows and took about twenty four man hours to create.

ARTIST

Briefly, ARTIST, by reading the description of the module, creates an initial internal representation of the layout to be generated. It then then manipulates the internal representation trying to reduce the layout's size. Finally, ARTIST generates the actual layout in a format compatible to either LEd or Magic from the final internal representation. ARTIST manipulates the internal representation by performing some number of trials. For each trial, ARTIST generates a new configuration of the internal representation, determines the layout size of the new configuration, and then possibly replaces the old configuration with the new configuration. Different versions of ARTIST can be characterized by how they generate successive new configurations and how they decide to replace an old configuration by a new configuration.

One of the most important decisions made during the design of ARTIST is concerned with how a mesh array is internally represented. It is generally more efficient to manipulate a high level internal representation (symbolic) than a low level internal

representation (paint rectangles). Hence, using a high level internal representation usually allows more configurations to be evaluated in a given period of time. However, since a low level description is usually tighter, it is less likely to hide possible optimizations and is at the smae time easier to use to determine the layout size of a configuration. One advantage of our overall approach is that the description itself (internally the description's parse tree) can be used almost without any information loss to almost directly represent the mesh array to be generated. This high level internal representation is particularly easy to manipulate. However, we can still efficiently determine the layout size for a given configuration.

One of the principal functions performed by ARTIST is the allocation of rows and columns of a mesh array to the subcircuits of the module. ARTIST performs this allocation using a module description in the following way. At the gate level, a mesh array can be view as illustrated in Figure 9.

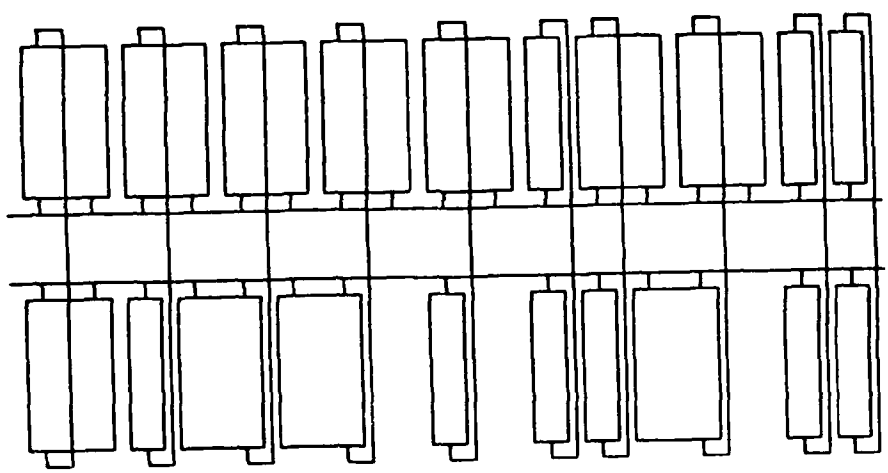


Figure 9. Block Diagram of mcell Layout

The pullup and pulldown part of each gate are allocated, respectively, consecutive columns in upper (N well) and lower (P well) areas of the array. Hence, the pullup (pulldown) parts of any two gates cannot overlap. Because of the second level metal wire connecting the two together, the areas allocated to the pullup and pulldown part of each gate must have at least one common column. These characteristics lead to the following observation. The order of the statement pairs associated with each of the defined variables can be used to specify the order of the gates and, consequently, the columns to be allocated to each pullup and pulldown.

The order of the operands for each operation supplies most of the remaining information needed by ARTIST. In the case of *&*, the row segments allocated to the left operand must precede the row segments allocated to the the right operand. In the case of |, the column segments allocated to the left operand must precede the column segments allocated to the right operand. The final information needed by ARTIST is how row segments are to be allocated. This information is not implied by the module description. layout to be generated.

For each well, ARTIST generates a layout row by row. For each row, row segments allocation and layout generation is performed as follows (although we have omited many details, the following overview is conceptually correct.)

1) Initially each of the expressions associated with the pullup part, if the N well area is being generated, or the pulldown part, if the P well area is being generated, of each gate is activated.
2) If the operation | is activated, then both of its operands are activated.
3) If the operation *&* is activated, then its left operand is activated.
4) If a variable is assigned to a segment row, it is deactivated.
5) If the left operand of the operation *&* is deactivated, the the right operand is activated.
6) If both operands of an operation is deactivated, then the operation is deactivated.
7) Generation of the layout for that well terminates when all operations and operands have been deactivated.

Before each row is allocated, ARTIST scans the row for variables which are active. Based on this information, ARTIST allocates row segments to some of the active variables, generates the layout for that row, and then deactivates the variables which were assigned. If an active variable cannot be assign to a row segment, the circuit associated with that variable is stretched into the next row.

While many row allocation algorithms have been tried (e.g., First Fit, Most Fit, FIFO), one conclusion seems clear. Any reasonable row allocation algorithm appears to work as good as any other. This seems to be the result of having to deal with very few active variables (as compared to the total number of variables) at each row.

ARTIST is being used on a regular basis at Penn State. Figure 10 illustrates the mesh array layout generated by ARTIST for the formal description of the mcell module given in Figure 2.

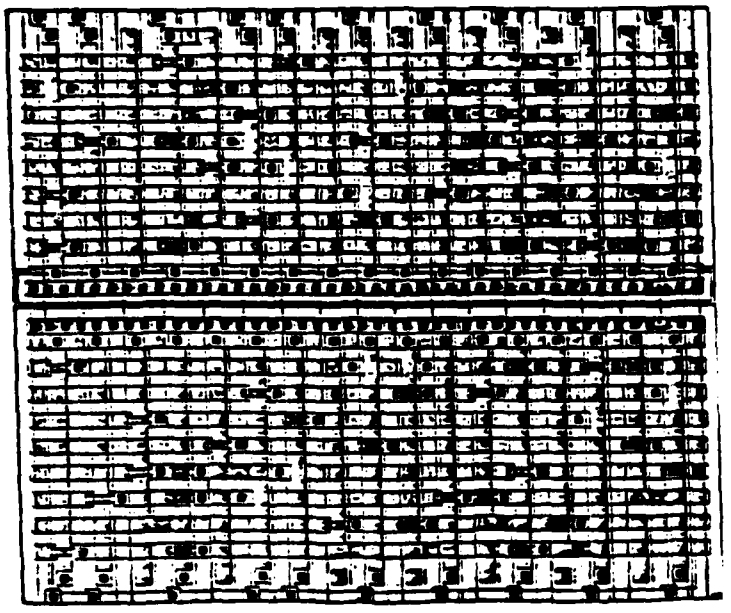


Figure 10. ARTIST Generated mcell Layout

This layout is seventeen columns by eighteen rows and took about one minute to create. Note that ARTIST created a smaller layout in far less time than the hand created layout. In creating the layout given in Figure 10, ARTIST performed about forty trials per second on a 68020 based workstation, a VALID Logic SCALDStar. Hence, the layout was generated using about two and half thousand trials. Doubling the complexity of the layout to be created approximately halfs the number of trials which can be performed per second and about twice as many trials must be performed to obtain a layout of similar size optimality.

LAYOUT OPTIMIZATIONS

While in a state of constant refinement, ARTIST is in use at Penn State for ongoing architecture research and class projects in VLSI courses. However, as pointed out in the introduction, ARTIST owes its existence to academic reasons as well as pragmatic reasons. Toward fulfilling the academic reasons, we are experimenting with several different versions of the approximation algorithms used to manipulate the internal description. These algorithms generate the new trial configuration by switching the order of statement and operand pairs. While our results are preliminary, they are interest-

ing.

We first wanted to develop a base line approximation algorithm by which other algorithms could be judged. The natural candidate for such an algorithm seemed to us to be a totally random algorithm. This algorithm, our Monte Carlo algorithm, uses a totally random configuration for each new trial. We then implemented what we thought would be the tried and true approximation algorithm, simulating annealing. Simulating annealing generates a new trial configuration by incrementally changing the old configuration. That is changing the order of only one statement or operand pair. Figure 11 gives a comparison of the layout obtained by the simulating annealing and Monte Carlo algorithms for a typical module of two hundred transistors.

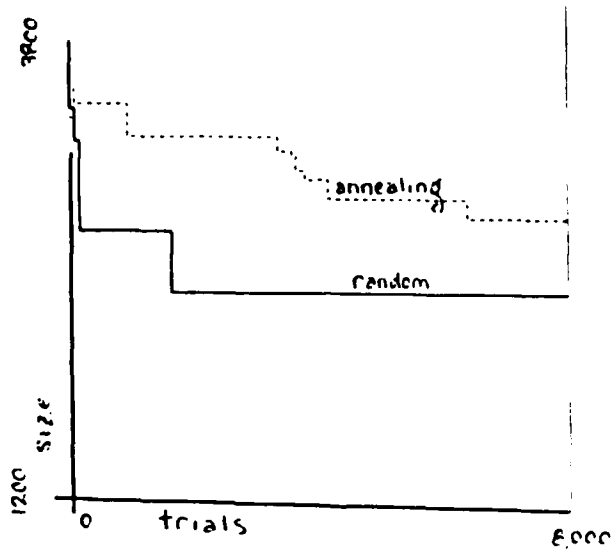


Figure 11. Random v.s. Annealing

Each line of Figure 11 represents the best configuration found using the indicated number of trials. Much to our surprise, for ten thousand trial configurations the random algorithm was better. Furthermore, this result could be consistently reproduced for different modules and cooling scheduling strategies.

We offer the following analogy as an explanation for this phenomenon. Suppose a person was standing, blindfolded, in a rectangularly tiled room. The person is trying to find the smoothest tile in the room. The person may move from tile to tile. After moving to a tile, the person may reach down and touch the tile to determine it's smoothness.

If the smoothness of a given tile is totally independent of the smoothness of the other tiles in the room, random walking (each move must be between adjacent tiles) or random leaps (each move need not be between adjacent tiles) is as good as any non-determistic strategy. Annealing succeeds when the smoothness of a given tile is not independent of the smoothness of the other tiles in the room. Using annealing, a person would compare the tile they are standing on with one of the adjacent tiles. They would then tend to move to the smoother of the two.

However, the very reason (dependence) that seems to make annealing work can produce very long search times. For example, suppose the tiles in one quadrant of the room have nearly the same smoothness and that the tiles in the other three quadrants are all much rougher than the tiles in the first quadrant and that many local optimums exist. Now start the person searching as far from the smooth quadrant as possible. To find the smooth quadrant, the person may (and probably does) spend a lot of time stumbling around the other three quadrants, since they can in effect see only the smoothness of the tiles in their immediate vicinity. However, if the person can leap randomly around the room, he would find one of the tiles in the smooth quadrant after only sixteen leaps with reasonably high probability. While our example seems contrived, it reflects to a remarkingly high degree the search space as seen by ARTIST - a few (as compared to the entire search space) relatively large global optimums (as compared to the entire search space) surrounded by may small local optimums.

The avocates of annealing would point out that after enough tries, the nonleaping person would find the smooth quadrant and would then find a solution even better than the random leaping person would have been able to find after the same number of tries. While we don't dispute this point, we only offer the observation that many hours of stumbling around may be necessary to reach this point.

Our first attempt to try to improve the efficiency of annealing was to use random initial trials and then to switch over to simulating annealing. The problem with this approach was in developing a good mechanism to determine when the switch between the two modes should take place. Our second attempt solves this problem. Simulated annealing is used through out the running of the algorithm. However, at the beginning, to generate a new configuration ARTIST makes n random incremental changes to the old configuration. Hence, if n is large enough, ARTIST evaluates almost random configurations. As ARTIST progresses, n is made smaller until it is only 1 (incremental configuration changes). Using initial values of 5 and 10 for n, we obtained the results

given in Figure 12 for same module used to obtain the results given in Figure 11.

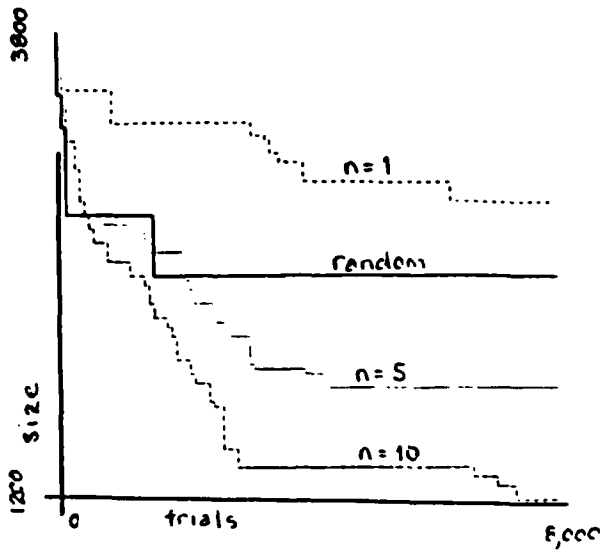


Figure 11. Random Annealing.

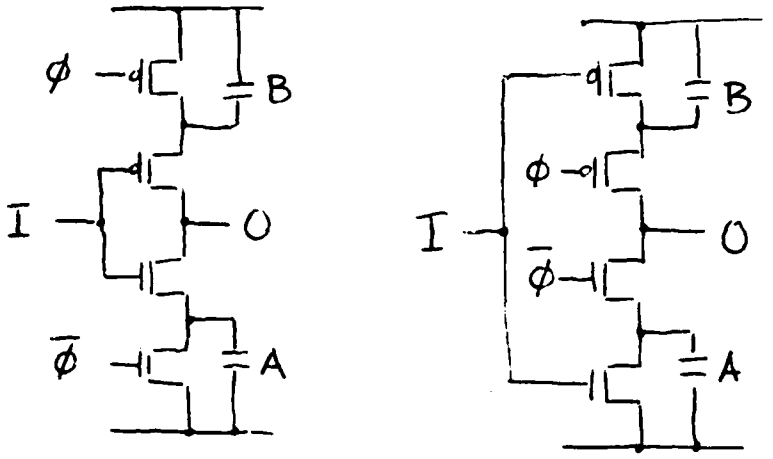
Again, each line of Figure 11 represents the best configuration found using the indicated number of trials. The results show that the hybrid algorithm is better than either a solely random or solely annealing approach. The best results were obtained for n equal to 10. These results could be consistently reproduced for different modules.

FUTURE DIRECTIONS

Our investigation into this area is actually far from over as we must now deal with two cooling schedules. However, the control of randomness does appear to be the more critical issue. We plan to further analyze the behavior of our current hybrid algorithm and to try other algorithms. We plan to expand ARTIST so that it can handle stacks of mesh arrays (several mesh arrays stacked one on top of the other) and to improve the performance of ARTIST so larger layouts can be handled.

At present, ARTIST assumes that logically equivalent circuits are likewise physically equivalent (produce the same behavior). This is not always the case. Consider, for example, charge sharing. Because of charge sharing, the following two circuits are not

physically equivalent because of effects due to the capacitors at *A* and *B*.



We plan to continue to investigate this problem.

ACKNOWLEDGEMENTS

This work has been supported in part by the Army Research Office under Contract DAAG29-83-K-0126. The acquisition of two VALID SCALDstar CAD workstations on which the tools are being developed was made possible by a grant from VALID Logic, Incorporated.

BIBLIOGRAPHY

Be Beekman, J., "Mesh Arrays for CMOS Circuit Design," Computer Science M.S. Thesis, PSU, August 1986.

BOI Beekman, J., R.M. Owens, and M.J. Irwin, "Mesh Arrays and LOGICIAN: A Tool for their Efficient Generation," submitted to DAC 87.

DR Denyer, P. and D. Renshaw, *VLSI Signal Processing: A Bit-Serial Approach*, Addison-Wesley, 1985.

IO Irwin, M.J. and R.M. Owens, "Digit Pipelined Arithmetic: A Tutorial," to appear in *Computer*.

KGV Kirkpatrick, S., C.D. Gelatt, M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, pp. 671-680, May 1983.

OI1 Owens, R.M. and M.J. Irwin, "An Overview of the Penn State Design System," submitted to DAC 87.

OI Owens, R.M. and M.J. Irwin, "A System for Designing, Simulating, and Testing High Performance VLSI Signal Processors," *IEEE Trans on CAD,* CAD-5(3), pp 420-428, July 1986.

Ow Owens, R.M., "YAHDL: Yet Another Hardware Description Language," Department of Computer Science Tech Report, Pennsylvania State University, University Park, Pa, 1986.

RI2 Reeves, D.S. and M.J. Irwin, "Switch-Level Verification of MOS Signal Processing Circuits," submitted to DAC 87.

RS Romeo, F. and A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications," *Proc. of the 1985 Chapel Hill Conf. on VLSI,* pp 393-419, NC, May 1985.

Ul Ullman, J. D. *Computational Aspects of VLSI,* Computer Science Press, 1984.

END

10-81

DTIC